An Experimental Comparison of Fault and Error Injection

J. Christmansson†, M. Hiller††, M. Rimén†

† Carlstedt Research & Technology AB
Stora Badhusgatan 18-20
S-411 21 Göteborg, Sweden

†† Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

The complex interactions between faults, errors, failures and fault handling mechanisms can be studied via injection experiments. This paper presents an investigation of both fault and error injection techniques for emulating software faults. For evaluation, 1600 software faults and 5400 time-triggered errors were injected into an embedded real-time system. The cost-related results are: (i) the time required to create a fault set for fault injection was about 120 times longer than the time required to create an error set for time-triggered injection and (ii) the execution time for the time-triggered error injection experiments were four times shorther than for the fault injection experiments. However, the error injection would be only 1.3 times faster if another strategy for fault injection had been used. Furthermore, failure symptom related results are: (i) the test case had a greater influence than the fault type on the failure symptoms for fault injections, (ii) the error type had a greater influence on the failure symptom for time-triggered error injections than had the test case and (iii) the error type had a larger impact on the failure symptoms than the fault type.

1. Introduction

Computers are currently employed to control applications such as nuclear power plants, aircraft and automobiles. The control of vehicle dynamic functions in future automobiles, for instance, will be totally dependent on computers, as these systems will be built without any mechanical backup devices. A failure in a computer system that controls such an application can thus lead to significant economic losses or even loss of human lives. Obviously, such a system must undergo a rigorous dependability validation and verification.

Fault injection is an attractive approach to the experimental dependability validation of fault-tolerant systems (see e.g. [1], [2]), as it provides the means for a detailed study of the complex interaction between faults, errors, failures and fault-handling mechanisms (Figure 1.1). Dependability validation of fault-tolerant systems by fault injection addresses fault removal and fault forecasting [3]. In the case of fault removal, fault injection uncovers potential fault tolerance deficiencies, e.g. transitions 3, 8 and 9 in Figure 1.1. In the case of fault forecasting, fault injection allows an estimation of the coverage factors, which are important parameters in analytical dependability models.

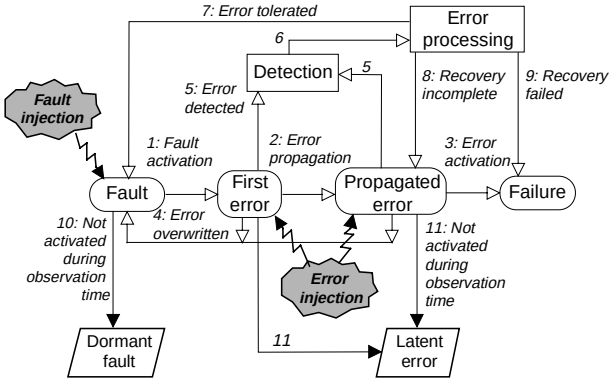

Figure 1.1. The fault, error and failure process in the context of fault and error injection.

In both the academic community and industry, most fault injection studies have been aimed at studying the effects of physical hardware faults, i.e. faults caused by wear-out or external disturbances (see e.g. [4], [5]).

Only a few studies have been concerned with injection and emulation of software faults, e.g. [6], [7]. It is often pointed out that SWIFI (SoftWare-Implemented Fault Injection) can be used to emulate software faults by injecting errors, but to the authors knowledge no studies have been published that compare the failure symptoms obtained using source code faults and SWIFI injected errors. SWIFI is done by changing the contents of memory or registers, based on some fault models, to emulate the occurrence of hardware or software faults.

This paper presents an investigation of both fault and error injection techniques for emulating software faults. More

specifically, the investigation focused on the following two aspects: (i) the cost in terms of setup and execution time for using the techniques and (ii) the impact of the test case, fault type and error type on the failure symptoms of the target system. We did not study coverage, as the target system did not have any fault handling mechanisms.

A Fault Injection Campaign Control Computer (FIC^3) was developed for this investigation. The FIC^3 injection environment is designed to inject faults and errors into an embedded real-time system using SWFI. Several papers on fault injection environments that use SWIFI have been published: FAUST [7], FIAT [8], FERRARI [9], HYBRID [10], FINE [6], FTAPE [11], XCEPTION [12], DOCTOR [13]. FIC^3 allows the injection of faults using modification of the source code, resembling the method used by FAUST. The error injection methods used by FIC^3 are based on modification of the memory, similar to the method used by FIAT, and processor registers, comparable to the methods used by FINE and FERRARI.

More specifically, the injection techniques supported by FIC^3 are (see Figure 1.1):

- fault injection (FI), i.e. a modified object file is loaded to the target system;
- event-triggered error injection (EIE), i.e. an error (first error) is injected at a breakpoint with the purpose to exactly mimic a software fault; and
- time-triggered error injection (EIT) which injects a propagated error at a certain point in time (in some cases even periodically).

Consequently, the FIC^3 environment can be used to emulate both software faults and intermittent hardware faults, since a propagated error can be caused by both.

As seen in Figure 1.1, an activation of a fault (transition 1) will cause the first error, and all events between the first error and a possible system failure (transition 3) are seen as error propagation (transition 2). The first error is defined as:

A software fault is activated and manifests itself as the first error when a processor register is loaded with an incorrect value.

This paper is organized as follows. Section 2 gives a user's view of the FIC^3 injection system, section 3 provides a case study describing experiments carried out on an aircraft arresting system. Section 4 contains a discussion on the FIC^3 and the obtained results, and section 5 presents a summary.

2. The Fault Injection Campaign Control Computer (FIC^3)

An injection campaign can be seen as three consecutive phases: a *setup phase*, an *injection phase* and an *analysis phase*. These phases are depicted in Figure 2.1.

The *setup phase* uses external data (e.g. source code, field defect distributions, usage profiles) to produce a campaign specification. The campaign specification defines the fault and/or the error sets, test cases and readouts that are to be collected from a target system. Several probes inserted into the target system do the readout collection; this manual instrumentation of the target system must be done on the basis of the readout specification given in the campaign specification.

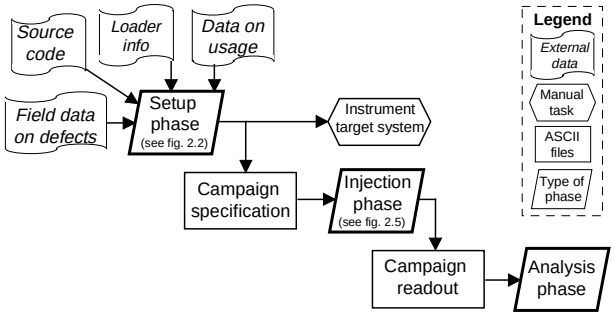

Figure 2.1. An overview of the phases in an injection campaign.

The campaign specification is used to automatically conduct an injection campaign during the *injection phase*. That is, faults and/or errors are injected into the target system while the test cases are executed and readouts are collected. The readouts collected during the injection campaign are stored in the campaign readout files. Relevant readouts are extracted from the campaign readout files during the *analysis phase*. The extracted readouts are then analyzed, and measures are computed, e.g. asymptotic error detection coverage. A more detailed description of these three phases is given in the sections below.

2.1 The setup phase

The setup phase consists of three main tasks (see Figure 2.2): generation of faults (F*) or errors (E*) according to defect distributions observed in the field; generation of test cases (U*) based on data on possible usage; and campaign setup, which creates a campaign specification for the injection experiments. Each injection experiment is defined by an experiment specification with four main components: a reference to an object load file for faults (f), parameters to an interrupt routine for errors (e), specification of a test case (u) and a specification of the readouts that are to be collected during the experiment.

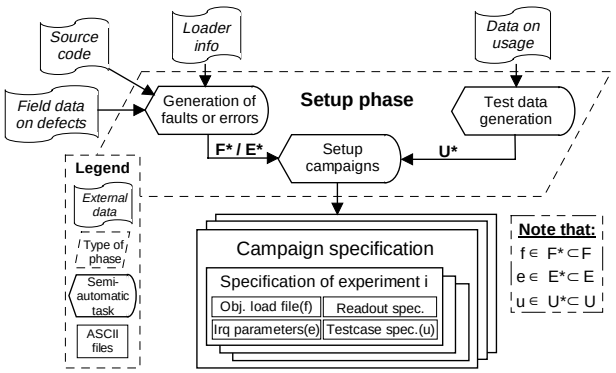

Figure 2.2. The setup phase: inputs required, tasks and produced output.

Generation of faults and errors. The *generation of faults* (F*) is currently a semiautomatic task, i.e. one sub-task requires manual work while the other sub-tasks are automatic. However, the *generation of errors* (E^*_{EIE}) is a manual task, as the FIC^3 system does not currently provide any automatic support. Thus, errors must be identified manually using loader information and source code listings. Note that the fault set targets software faults, whereas the error set can mimic both software faults and intermittent hardware faults.

Generation of FI faults , i.e. the generation of modified load files, is done via four sub-tasks (see Figure 2.3): identification of possible faults, selection of faults via random sampling, manual insertion of the sampled faults and automatic generation of object code load files. The generation of EIE errors, i.e. the identification of error parameters to the interrupt routine (see Figure 2.3), is a manual task.

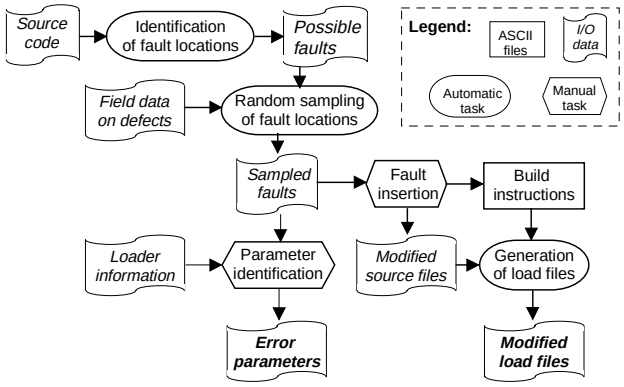

Figure 2.3. The generation of a fault set (F*) or an error set (E^*_{EIE}).

The generation of the fault set (F*) starts with an automatic identification of possible fault locations. That is, a parser will use the target systems source code to automatically generate a list of possible software faults (F_p). This list of possible faults is actually a subset of all possible faults (F) in the target system, i.e. $F_p \subset F$. Nevertheless, the size of F_p can be too large to use exhaustively. Therefore, the list of faults that are to be injected is selected by means of random sampling from F_p. This sampling can be done according to any distribution, but the preference is a distribution obtained from the field. The selected faults must then be manually inserted into the source code, and each fault is saved as a modified source file. However, it should be possible to automate the fault insertion. All the modified source files are then automatically recompiled and re-linked to object load files (F*). This automatic generation of load files is governed by a set of build instructions. The set of selected faults thus consists of a number of modified load files, one file for each fault (f). Note that: $f \in F^* \subset F_p \subset F$.

The generation of an error set (E^*_{EIE}) for event-triggered injection is based on the list of selected faults and on loader information. The rationale for this is that we want to mimic software faults via the injection of errors. That is, a software fault is mimicked via injection of the first error (see Figure 1.1). The error injection can be done via modification of: a register content, a memory cell or a control flow. The event-triggered injection is used to ensure that the first error is injected, i.e. the injection is synchronized with the execution of the fault location. The decision as to *when* (i.e. synchronized with an address in the text segment) to inject *what* (e.g. incorrect value) *where* (e.g. into register D) is made on the basis of source code and loader information. The answers to *when*, *what* and *where* provide the parameters for an error injection routine. Thus, the set of selected errors consists of error injector parameters, one file for each error (e). Note that: $e \in E^*_{EIE} \subset E$.

Time-triggered, i.e. clock activated, error injection is used to mimic a propagated error. For EIT, the answers to *when*, *what* and *where* are selected without considering the source code statements. The generation of an error set (E^*_{EIT}) for time-triggered injection might use loader information if a stack or global variables are to be targeted.

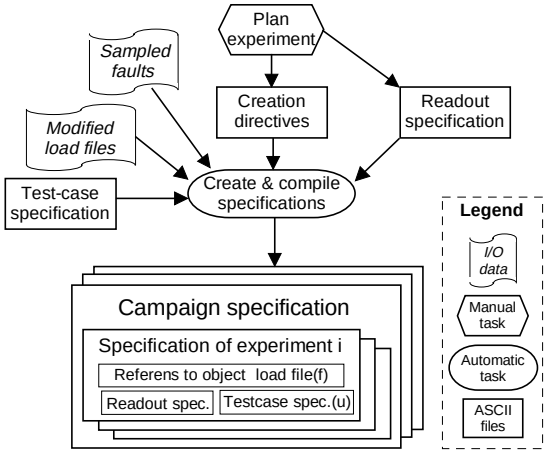

Figure 2.4. The generation of campaign specifications for fault injection.

Campaign setup. The *campaign setup* is a semiautomatic task, i.e. the first sub-task requires manual work and the second sub-task is automatic. The two tasks to be performed in campaign setup are (see Figure 2.4): experimental planning and automatic generation and compilation of specifications.

The experimental planning involves: deciding which measures to obtain from the campaign, identifying the readouts used to calculate the measures and creating a readout specification defining the readouts to be collected.

2.2 The injection phase

The *injection phase* will conduct the actual injection campaign, and each injection experiment consists of five main automatic tasks that are driven by the campaign specification. The tasks carried out during one experiment are (see Figure 2.5): reset the target system, download a fault (f) or an error (e); trigger the error injection; control the environment simulators; and control the readout collection.

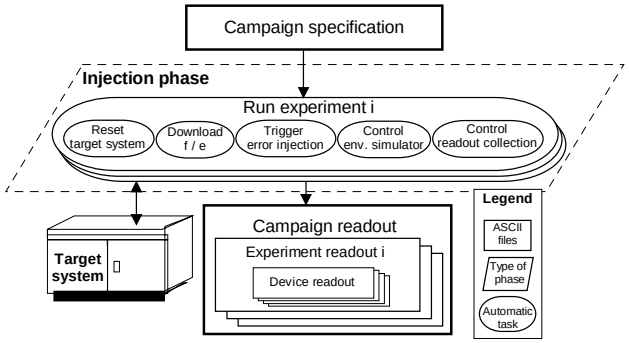

Figure 2.5. The injection phase: input required, tasks, the affected target system and collected readouts.

The reset starts the application in the target system from a "clean state". The internal state of the target system might be corrupted during the previous experiment, and the reset is therefore required to ensure independence between the experiments.

The faults and errors are downloaded to the target system via a serial link. The injection type defined by the experiment specification decides whether a fault or an error shall be downloaded. The injection types are: *golden run*, *fault injection* or *error injection*. A *golden run* means that the correct object load file is downloaded to the target system. A fault injection implies that a modified object load file is sent to the target system, whereas an error injection calls for the transmission of error injection parameters - which are given by the experiment specification - to an interrupt routine. The interrupt routine will inject an error into the target system, i.e. modify the internal system state. This interrupt routine will be activated by a trigger task.

The trigger task will activate the error-injecting interrupt routine when the injection condition is satisfied. That is, an interrupt will be generated either when the emulated fault location is executed (EIE) or at a particular time (EIT).

The environment simulator provides an artificial environment to the target system. The simulator must thus mimic the environments response to actions taken by the target system. Furthermore, the simulator must provide artificial environment stimuli to the target system. Thus, a system that interfaces with the environment at N number of points calls for N tailored simulators. These simulators are controlled in a standardized and general way by means of the test case specifications.

The readout collection is done by the environment simulators, which log the output that the system sends to the environment. Furthermore, special hardware probes are inserted into the target system, and dedicated software control the readout collection. A device readout file is created for each probe and for each environment simulator. References to these device readout files are collected and stored in an experiment readout file, and references to the experiment readout files are in turn stored in the campaign readout file.

2.3 The analysis phase

The *analysis phase* extracts and analyzes the readouts stored in the readout files. This is done in three main steps: extract relevant events from the device readout files, interpret the event data and compute required measures. The task carried out during this phase is often quite specific for the experiment at hand, and this phase will not be further discussed.

3. Case study

A case study was carried out with the purpose of investigating both fault and error injection techniques for emulating software faults. Fault injection and error injection experiments targeting an embedded real-time system were conducted.

Table 3.1. Summary of the injection experiments.

Type	Technique	Measurements	Injections
FI	Modification of source code	Setup cost Execution cost Distribution of failure symptoms	1600
EIE	Breakpoint activated interrupt routine	Setup cost	0
EIT	Clock activated interrupt routine	Setup cost Execution cost Distribution of failure symptoms	5400

The following two types of experiments were conducted: fault injection (FI) and error injection time-triggered (EIT). Error injection event-triggered (EIE) experiments were setup but not conducted. The assumption is that EIE will result in exact emulation of a software fault and that deviations would be the cause of faulty parameters or defects in the FIC^3 error injection module. Two types of measurements were collected: cost in terms of time and the distribution of failure symptoms. The time needed to create the fault or the error set was measured during the Setup phase. The creation of the sets for FI, EIE and EIT were carried out by three different persons, i.e. each author created one set. The time it took to conduct the FI and EIT experiments was also measured. However, the time needed to build the supporting software tools was not logged.

These measurements are described in the text below, and the experiments conducted are summarized in Table 3.1 above.

3.1 Experimental setup

Target system. The target system is designed to arrest aircraft landing on a runway. A fighter equipped with a hook is stopped by means of a cable barrier (i.e. as on an aircraft carrier). The requirements on the cable barrier can be found in [14]. The barrier consists - besides the physical barrier hardware - of:

- Two tape drums, one on each side of the runway, connected via a cable.
- A Programmable Logic Controller (PLC) which controls all functions related to the physical barrier, e.g. raising the barrier, lowering the barrier and monitoring the hydraulic brake system.
- A master node, which periodically reads the rotation sensor and starts an arrest when the sensor indicates rotation on its tape drum. The master node uses the rotation sensor to measure the speed of the landing aircraft and controls a brake pressure valve using a software-implemented PID regulator. This valve feeds hydraulic oil to the brake cylinder, which is used to slow the rotation of one of the tape drums. Furthermore, the master node sends the desired brake pressure to the slave node, which controls the pressure valve on the other tape drum.
- A slave node, which is basically a software-implemented PID regulator that controls the brake pressure valve, i.e. it reads the pressure sensor and checks whether it conforms to the pressure ordered by the master node.

The setup of the experimental environment can be seen in Figure 3.1. The physical barrier hardware and the landing aircraft is simulated, i.e. the master node and the slave node are communicating with simulated sensors and actuators. The injection experiments target the master node.

The simulator is driven by the FIC^3 environment (for details see [15]). This setup enables us to automatically inject faults and/or errors while the target system is subjected to different test cases. A test case is defined by the aircraft weight (kg) and its engagement speed (m/s).

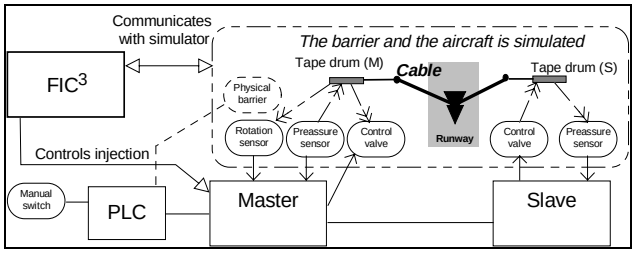

Figure 3.1. The target system; an aircraft arresting system.

Generation of a fault set. The targeted processor has no operating system, and the targeted part of the application (16 KB object code) is written in C. The C Fault Locator (CFL, see [15] for details) identified 991 possible locations ($F_p \subset F$) of the fault types assignment (A), checking (C) and interface (I), which are a subset of the fault types of the ODC, Orthogonal Defect Classification [16]. The resulting distribution is shown in column four in Table 3.2.

Table 3.2. Distribution of the 991 possible faults and of the 200 sampled faults.

Fault type	Possible fault		Sampled faults	
Assignment	514	52%	92	46%
Checking	223	23%	46	23%
Interface	254	26%	62	31%
Total	991	100%	200	100%

To reduce the labor in the setup phase, it was decided that 200 faults would be injected. A random sampling of fault locations was used to sample (without replacement) 200 faults from the list of possible faults generated by the CFL program. As we had no a priori knowledge of the distribution of software faults in the target system we were forced to sample the 200 faults according to a uniform distribution. The outcome of that sampling is shown in column 6 in Table 3.2.

The 200 sampled faults (F^*) were manually inserted into the source code. One load file was generated for each inserted fault (f). Recall that $f \in F^* \subset F_p \subset F$.

Generation of an error set for EIE. EIE is supposed to exactly mimic the behavior of software faults. The technique used in EIE, is to force the processor into making incorrect computations, e.g. computing the address for

`p[i]` instead of the address of `t[i]` (see [15] for details). Predefined error injection routines are used to manipulate the processor. Consequently, the generation of an error set for EIE involves identification of parameters to these routines. We attempted to identify parameters for an error set which would exactly mimic the behavior of the 200 faults mentioned above (F*). However, we were not able to create error injector parameters for 15 of the faults. The main problems during identification of the parameters involved: off line location of an intermediate value stored on the stack; and modification of comparison operators in a Boolean expression, e.g. forcing the processor to check for "`(A < B) && !done`" instead of "`(A >= B) && !done`".

Generation of an error set for EIT. A set of 675 errors aimed at EIT was designed to both mimic software faults and intermittent hardware faults. Periodic EIT, with the same period as the main control loop in the target system, was employed as an attempt to mimic software faults. The following types of errors were injected:

- Modification of memory (EIT (Mem))
  - 50 bit-flips, each in a randomly selected bit and byte of the stack area, and
  - 225 bit-flips, each in a randomly selected bit and byte of the global data area. That is, 25 errors were injected in the global data area in each of the nine source code modules of the target system;
- Modification of a register (EIT (Reg))
  - 200 bit-flips, each in a randomly selected bit in a 16-bit register, i.e. 50 errors in PC, D, X and Y, respectively, and
  - 50 bit-flips in randomly selected bit in the control code register; and
- 150 delays of the processor execution (EIT (Del)). That is, 50 delays for each of the times 0.4, 1.6 and 4.0 ms account for 5%, 20% and 50% of the main loop period, respectively.

These errors were injected every eight milliseconds, the first injection time is randomly selected within the period. The rest of the injections are periodical during the observation time of 40 seconds, resulting in 5000 activations of the error injector for each injection run. Each activation of the error injector requires 167 execution cycles of the Motorola M68HC11, which corresponds to 83.5 microseconds. Hence, the total execution time for the error injector is 0.42 seconds, which means that the injector "steals" approximately 1% of the observation time. This is also the case for the golden run, but the error injector executes only 167 NOPs, i.e. no error is injected.

Test cases. The main function of the target system is to arrest an incoming airplane. The barrier and the airplane are simulated. The following parameters control the simulator:

- engaging speed: 30 m/s to 100 m/s
- aircraft weight: 4000 kg to 25 000 kg

Eight test cases corresponding to three types of aircraft with three or two different engaging speeds were manually generated. The test cases and the resulting kinetic energy (i.e. $E_k = (mv^2) / 2$) are shown in Table 3.3.

Table 3.3. The eight test cases.

		Test cases (tc)								
Aircraft	M (kg)	tc #	v (m/s)	E_k (kJ)	Tc #	v (m/s)	E_k (kJ)	tc #	v (m/s)	E_k (kJ)
S18	23300	1	50	18640	2	70	41940	3	80	74560
F16	16000	4	50	20000	5	70	39200	6	80	51200
A4	12400	7	40	15500	8	60	30380			

The eight test cases shown in Table 3.3 were grouped into three categories. This was done on the basis of the kinetic energy level. Three energy levels were considered: low, medium and high. The three levels and the related test case number are shown in Table 3.4.

Table 3.4. The three levels of kinetic energy.

Energy level	Range (MJ)	Test case
High (Hi)	45 - 75	tc 3, tc 6
Medium (Med)	30 - 45	tc2, tc 5, tc 8
Low (Low)	15 - 30	tc 1, tc 4, tc 7

Setup cost. The Setup time is given by (1) and is composed of: the time required to design a fault or an error set with N elements and the time needed for the creation of M test cases.

$$T_{setup} = \sum_{i=1}^{N} (T_{Ii} + T_{Ci} + T_{Bi}) + \sum_{j=1}^{M} (T_{Ii} + T_{Ci}) \quad (1)$$

In (1), T_I is the time needed to identify, T_C the time needed to create and T_B the time required to automatically build the fault or error set. T_I and T_C apply for both the fault/error set and for the set of test cases. We will measure only times related to the generation of the fault or the error set.

Execution cost. The time required to conduct an injection campaign is given by (2). This approach was used to carry out the experiments presented in this paper.

$$T_{exe_1} = \sum_{j=1}^{M} \sum_{i=1}^{N} (T_{Lij} + T_{Sij} + T_{Oij}) \quad (2)$$

In (2), T_L is the time needed to load a fault or an error, T_S the synchronization time and T_O the observation time. The second summation (or inner loop) represents the N elements in the fault or the error set, while the first summation is related to the M test cases. With a minor alteration,

the load time (T_L) does not need to be part of both summations, which is shown by (3).

$$T_{exe_2} = \sum_{i=1}^{N}(T_{Li} + \sum_{j=1}^{M}(T_{Sij} + T_{Oij})) \quad (3)$$

Consequently, injection campaigns can be conducted in two ways: (i) download a new fault for each injection and (ii) download a new fault for each test case. The execution time required by the first approach is given by (2), and the time needed for the second is given by (3).

Classification of failure symptoms. The readout files from the aircraft simulator were used to identify the various failure symptoms. The following parameters were used: Retardation force (R) on hook; brake force (B) on cable; position (p) of aircraft; brake pressure ordered by master (P_{master}); and brake pressure ordered by slave (P_{slave}).

The failure indicators considered are in order of criticality:

1. *Sliding hook* (SL_HOOK). The absolute error between the perpendicular brake force from the master and the perpendicular brake force from the slave is larger then the friction force ($\mu_{stat} * R$).
2. *Runaway* (RUNAWAY). The aircraft passes the maximum runway position (i.e. p >335 m).
3. *Incorrect stop position* (IN_S_POS). The aircraft stop position deviates more than 20 m from the stop position measured during the golden run.
4. *Incorrect pressure* (IN_PRESS). The absolute error between the golden run and the experiment run (e_{avg}) exceeds a predefined limit.
5. *Other failure* (Other). This includes two types: the simulator crashes as the target system behaves strangely due to an injection; and the retardation force exceeds 2.8 g times the mass of the aircraft.

An injection run that does not result in any failure indicators is classified as *dormant* (DORMANT).

3.2 Results

Cost measures. The average time needed to identify and create a fault or an error is shown in Table 3.5. This table also shows the average time required to build, i.e. compile and link an object load file. The build average of 1.07 seconds does not include the time required to correct nine compilation errors that were caused by incorrect creation of these nine faults. Furthermore, these average times for identification do not include the time required to sample the fault set. However, the sampling times are the same for EIE and FI, as both use the exact same set of sampled faults.

The time required to generate the 675 errors used for EIT is negligible, as it took 2 minutes and 31 seconds to generate the parameters for the injection of these 675 errors. An interesting observation is that the average time required to identify and create a fault set for FI and an error set for EIE are comparable: 110 seconds and 115 seconds for FI and EIE, respectively. This is quite surprising, as we expected that the error set generation would take much longer time.

Table 3.5. Average Setup cost for the different injection types

Seconds in cell	FI	EIE	EIT
Average identification time (T_I)	53	78	≈0.22
Average create time (T_C)	57	37	≈0.005
Average build time (T_B)	1.07	n/a	n/a

The average times required to load, synchronize and observe during the injection experiments are shown in Table 3.6.

Table 3.6. Average execution cost for the different injection types

Seconds in cell	FI	EIE	EIT
Average load time (T_L)	191	n/a	2.8
Average synchronization time (T_S)	20	n/a	20
Average Observation time (T_O)	40	n/a	40
Total	251	n/a	62.8

Failure symptoms. The distribution of the failure symptoms is shown for all fault injections (FI (All)), assignment faults (FI (A)), checking faults (FI (C)), interface faults (FI (I)), all error injections (EIT (All)), modification of memory (EIT (Mem)), modification of registers (EIT (Reg)) and delay of the processor execution (EIT (Del)), in bars one through eight in Figure 3.2, respectively.

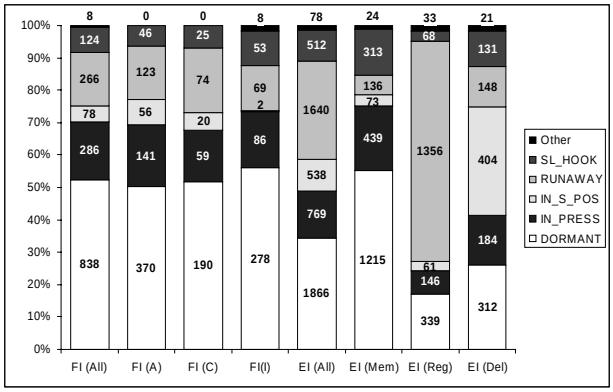

Figure 3.2. Failure symptoms by type of injection

Figure 3.2 shows the distribution of the failure symptoms as a function of the fault type (i.e. FI (A), FI (C) and FI (I)). It seems that the fault type does have an effect on the observed failure symptom. The fault type interface (I), for instance, seems to cause an unproportionally large frac-

tion of the failure symptom sliding hook (SL_HOOK). Moreover, the fault type checking (C) appears to cause a large proportion of runaway (RUNAWAY) failures. However, the question is: are these differences significant?

A simple non-parametric statistical test that can be used to determine the significance of the differences is the χ^2 test for k independent samples [17]. Let the null hypothesis be: *The failure symptoms are independent of the fault type*. The level of significance is set to 0.001, and the degrees of freedom are 10. Hence, the critical value of the chi-square at the 0.001 significance level ($\chi^2_{0.001}(10)$) equals 29.59. The chi-square value (q(10)) computed from the data equals 68.51, and the probability of occurrence for q(10) equals 2.5×10^{-10}. Consequently, the null hypothesis can be rejected ($\chi^2_{0.001}(10) = 29.59 < q(10) = 68.51$, $p = 2.5 \times 10^{-10}$).

Figure 3.2 also shows the distribution of the failure symptoms as a function of the error injection procedure used (i.e. EIT(Mem), EIT(Reg) and EIT(Del)). Clearly, the type of injection used has an impact on the failure symptom. The χ^2 test will be used to determine whether the differences are significant. Let the null hypothesis be: *The failure symptoms are independent of the type of injected error*. The null hypothesis can be rejected as: $\chi^2_{0.00001}(10) = 35.36 < q(10) = 3076.0$, $p < 10^{-300}$.

Figure 3.3 shows the distribution of the failure symptoms as a function of the type of test case (i.e. Low, Medium and High kinetic energy) for both FI and EIT. The test case type does have an effect on the observed failure symptom for both FI and EIT. Again the χ^2 test will be used to determine whether the differences are significant.

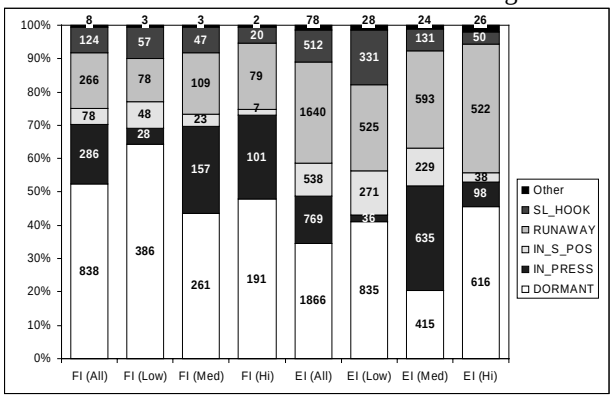

Figure 3.3. Failure symptoms by type of test case (energy level)

Let the null hypothesis be: *The failure symptoms for FI are independent of the type of test case*. The null hypothesis can be rejected as: $\chi^2_{0.0001}(10) = 35.56 < q(10) = 172.60$, $p = 1.75 \times 10^{-25}$.

Let the null hypothesis be: *The failure symptoms for EIT are independent of the type of test case*. The null hypothesis can be rejected as: $\chi^2_{0.00001}(10) = 35.56 < q(10) = 1185.5$, $p = 1.93 \times 10^{-248}$.

4. Discussion

4.1 Injection cost

Setup cost. The technique with the lowest demands on both setup time and execution time was EIT. FI and EIE were comparable in terms of Setup cost. The cost for FI could be drastically reduced if the manual fault insertion (see Figure 2.3) is automated. However, we see no simple way to automate the parameter generation for EIE.

Execution cost. The main reason for the large difference in execution time between FI and EIT can be attributed to the time required to download an object load file to the target processor, i.e. 3 minutes and 11 seconds. That time must be related to the observation time (i.e. 40 seconds) and the synchronization time (i.e. 20 seconds). Hence, a reduction in the number of downloads would reduce the execution time. This was discussed above, and formulas were provided for the computation of the execution time.

Table 4.1 shows a relative comparison between execution time for different injection techniques. This comparison uses the data presented in Table 3.6, and formula (2) and (3) from section 3.1. It is assumed in Table 4.1 that the fault and error set has 100 members and that two sets of test cases exist: one with 10 tests and the other with 100 tests. Then, the total number of experiments equal 1000 and 10000, respectively. The numbers shown in Table 4.1 are speedup in execution time as compared with FI according to formula (2).

Table 4.1. Execution time speedup

Speedup normalized to FI and formula (2)	10 test cases		100 test cases	
	FI	EIT	FI	EIT
Formula (2)	1.0	4.0	1.0	4.0
Formula (3)	3.2	4.2	4.1	4.2

Summary. The execution times are comparable for FI and EIT, however the setup cost is negligible for EIT. Note that the setup time mainly consists of manual labor, and can therefore not be compared with the execution time. Our low-cost choice would thus be EIT if low realism of the injected faults is acceptable.

4.2 Impact of fault type and test case

The previously presented results provide evidence that the fault type and the test case have an impact on the resulting failure symptom. A non-parametric statistical test - the χ^2 test for k independent samples - enabled us to conclude that there is a relationship between these variables. The main question is the strength of these relations, and a

simple indicator of the strength of a relationship exists. Let us, for instance, compute Pearson's contingency coefficient (Pcc) for the chi-square values computed above, see [18, p. 305]. A contingency coefficient is 0 for independent variables and $1/\sqrt{2}$ in case of a perfect relationship. The upper limit of the contingency coefficient can be adjusted to make the interpretation easier, i.e. 0 for independence and 1 for a perfect relationship. The type of injection, the null hypothesis, the probability of occurrence (p) for the computed chi-square value and Pearson's contingency coefficient are shown in Table 4.2.

Table 4.2. A summary of conducted tests

Type	Null hypothesis	p	Pcc	Pcc$*\sqrt{2}$
FI	The failure symptoms are independent of the fault type	$2.5x10^{-10}$	0.20	0.28
FI	The failure symptoms are independent of the test case	$1.75x10^{-25}$	0.30	0.42
EIT	The failure symptoms are independent of the error type	$< 10^{-300}$	0.60	0.85
EIT	The failure symptoms are independent of the test case	$1.93x10^{-248}$	0.42	0.60

Our interpretation of Pcc is:

- The fault type and test case had an impact on the failure symptoms in the case of FI, and the test case appears to have more influence on the symptoms.
- The injected error type and test case had an impact on the failure symptoms in the case of EIT, and the type of injected error seems to have the greatest influence of the two.
- The type of injected error had larger impact (0.85) on the failure symptoms, as compared to the type of injected fault (0.28). This can also be seen in Figure 3.2.

The test case had a large influence on the failure symptoms for both faults and errors, this indicates that a large amount of test cases should be applied in injection experiments. The great impact of the test case on the outcome of EIT experiments was not expected. The main cause for this impact is that the application program is better exercised for medium energy levels, since it periodically re-computes the desired pressure level. For low and high energy levels the application uses predefined pressure levels to a higher extent, and is therefore less sensitive to errors in calculated variables.

4.3 Emulating software faults

The most accurate way to emulate software faults is, obviously, to modify the source code (FI). Another approach that emulates software faults is to force the processor into incorrect behavior via modification of register and memory contents. Synchronizing the modification of the processor content with the execution of code (EIE) provides a technique that in most cases exactly mimics the behavior of simple, single statement software faults such as the ones used in this study. This injection technique can be especially useful for the injection of software faults targeting an embedded system whose text segment is stored in ROM. However, if a set of generic error injection routines is employed, all types of software faults may not be possible to emulate using EIE.

EIT emulates propagated errors, and it can therefore be used to mimic both software faults and intermittent hardware faults. However, EIT does not guarantee that a particular software fault is mimicked, nor does it enable us to emulate specific distributions of fault types. Nevertheless, EIT into memory cells storing global variables and stack data resulted in a distribution of failure symptoms that at first glance is quite similar to the outcome of FI and might therefore be possible to use as a low-setup-cost method to test a system.

To summarize our discussion, FI is the most accurate technique, followed by EIE, while EIT is the least accurate technique. Furthermore, both EIE and EIT may disturb the timing characteristics of the target system.

5. Summary

This paper presents an investigation of both fault and error injection techniques for emulating software faults. For evaluation, 1600 software faults and 5400 time-triggered errors were injected into an embedded real-time system. A Fault Injection Campaign Control Computer (FIC^3) was developed for this investigation.

Three types of fault and error sets were created: one set with 200 software faults for fault injection (FI); one set with 185 errors that were aimed at emulating the software faults (EIE); and one set with 675 errors that were meant to stress the system (EIT). Furthermore, two types of injection experiments were conducted: FI, which modified the source code and downloaded a new object file for each experiment, and EIT, which used a clock activated interrupt routine to inject an error periodically. Eight test cases, 200 faults and eight golden runs resulted in a total of 1608 FI experiments. Eight test cases, 675 errors and eight golden runs give a total of 5408 EIT experiments. From our investigations we can see that:

- FI is the most accurate technique, followed by EIE, while EIT is the least accurate technique.
- The execution times are comparable for FI and EIT, however the setup cost is negligible for EIT. Note that setup time mostly consists of manual labor, and can therefore not be compared with execution time. Our low-cost choice would thus be EIT if low realism of the injected faults is acceptable.

- The test case had a large influence on the failure symptoms for both faults and errors, which indicates that a large amount of test cases should be applied in injection experiments.

Acknowledgement

The authors wish to thank Robert Feldt and Susanne Bolin for taking the time to review early versions of this paper. We are also grateful for the insightful comments provided by the anonymous reviewers.

References

[1] R. Chillarege, N. S. Bowen, "Understanding Large System Failures - A Fault Injection Experiment", *Proc. 19th Int. Symp. On Fault Tolerant Computing*, pp. 356-363, June, 1989.

[2] R.K. Iyer, "Experimental Evaluation", *Special Issue FTCS-25 Silver Jubilee, 25th Int. Symp. on Fault Tolerant Computing*, pp. 115-132, June, 1995.

[3] J.C. Laprie (ed.), "Dependability: Basic Concepts and Terminology", *Dependable Computing and Fault-Tolerant Systems series*, Vol. 5, Springer-Verlag, 1992.

[4] J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications", *IEEE Trans. On Software Eng.*, vol. 16, no. 2, pp. 166-182, February, 1990.

[5] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, U. Gunneflo, "Using Heavy-ion Radiation to Validate Fault-Handling Mechanism", *Proc. Int. Test Conference*, pp. 140-149, 1991.

[6] W.L. Kao, "Experimental Study of Software Dependability", Ph.D. thesis, *Technical report CRHC-94-16*, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, 1994.

[7] J. Hudak, B.H. Suh, D. Siewiorek, Z. Segall, "Evaluation & Comparison of Fault-Tolerant Software Techniques", *IEEE Trans. on Reliability*, Vol. 42, No. 2, pp. 190-204, June 1993.

[8] Z. Segall, et al., "FIAT – Fault-Injection based Automated Testing environment", *Proc. 18th Int. Symp. On Fault Tolerant Computing*, pp. 102-107, June, 1988.

[9] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", *IEEE Trans. on Computers*, Vol. 44, No. 2, pp. 248-260, February, 1995.

[10] L. Young, R.K. Iyer, K. Goswami, and C. Alonso, "Hybrid Monitor Assisted Fault Injection Environment", *Proc. Third IFIP Working Conference on Dependable Computing for Critical Applications*, pp. 163-174, September, 1992.

[11] T.K. Tsai, R.K. Iyer, "An approach towards Benchmarking of Fault-Tolerant Commercial Systems", *Proc. 26th Int. Symp. On Fault Tolerant Computing*, pp. 314-325, June, 1996.

[12] J. Carreira, et al., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", *Proc. Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, pp. 135-149, September, 1995.

[13] S. Han, K.G. Shin, H.A. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-Time System", *Proc. IPDS'95*, pp. 204-213, 1995.

[14] US Air Force - 99, "Military specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction", *MIL-A-38202C*, Notice 1, US Department of Defense, September 2, 1986.

[15] J. Christmansson, M. Rimén, "A fault injection campaign control computer (FIC3)", *Technical report no. 298*, Chalmers University of Technology, Göteborg, Sweden, December, 1997.

[16] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Trans. on Software Eng.*, Vol. 18, No. 11, pp. 943-956, November, 1992.

[17] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 3rd ed., McGraw-Hill, ISBN 0-07-100870-5, 1991.

[18] H.M. Blalock, *Social Statistics*, 2nd ed., McGraw-Hill, ISBN 0-07-066175-8, 1979.